1. Tempo disponibile 120 minuti (90 minuti per gli studenti di "Introduzione agli Algoritmi" - 6 CFU, che devono fare solo i primi 3 esercizi).
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo Mystery:

Algorithm 1: Mystery(Int $A[1..n]$)

```
A.heapify()
i ← 1
while i ≤ n do
    print(A.findMax())
    A.deleteMax()
    i ← i + i
end
```

Soluzione. L'operazione **heapify** trasforma un array in un heap con costo lineare, quindi con costo computazionale $\Theta(n)$. Il ciclo **while** viene eseguito una quantità logaritmica di volte, in quanto ad ogni ciclo l'indice di controllo i viene raddoppiato. Le operazioni eseguite all'interno del ciclo sono a costo costante (**findMax** e aggiornamento dell'indice i) oppure logaritmiche nella dimensione dell'heap (**deleteMax**). La dimensione dell'heap è superiormente limitato da n, quindi possiamo quantificare il costo di ogni **deleteMax** con $O(\log n)$. Il costo complessivo del ciclo **while** è quindi $O(\log n \times \log n)$. Complessivamente abbiamo quindi il seguente costo computazionale per la funzione Mystery: $T(n) = \Theta(n) + O(\log n \times \log n)$. Abbiamo però che $\log n \times \log n = O(n)$; questo può essere compreso ricordando dalle slide del corso che $\log n = O(n^{\frac{1}{2}})$ e quindi anche $(\log n)^2 = O((n^{\frac{1}{2}})^2)$, da cui appunto segue $\log n \times \log n = O(n)$. Questo permette di concludere che $T(n) = \Theta(n)$.

2. Si scriva un algoritmo che preso in input un **albero binario di ricerca** T contenente chiavi intere non ripetute ed un intero $k \geq 1$, ritorni il k-esimo intero più piccolo contenuto nell'albero. Se T contiene meno di k chiavi, l'algoritmo ritorna NA (costante che indica valore non disponibile). Discutere la complessità della soluzione proposta.

Soluzione. Possiamo fornire diverse soluzioni al problema a seconda delle assunzioni che facciamo rispetto alla nostra struttura dati. Vediamo qui tre possibili approcci. Se assumiamo che i nodi del nostro albero binario di ricerca mantengano come informazione il numero di nodi nel proprio sottoalbero sinistro, una implementazione particolarmente efficiente è la seguente (Algoritmo 2).

Per individuare il k-esimo intero più piccolo, l'algoritmo 2 effettua una visita su un percorso radice-foglia. Nel caso peggiore tale visita avrà una lunghezza pari all'altezza h dell'albero, quindi O(h).

Algorithm 2: kthSmallestKey1(BST T, Int k) → Int

```
if T = NULL then
|  return NA
else if k = T.leftNodes +1 then
|  return T.key
else if k < T.leftNodes then
|  return kthSmallestKey1(T.left,k)
else
|  return kthSmallestKey1(T.right,k-T.leftNodes-1)
end
```

Come seconda possibilità notiamo che in un albero binario di ricerca è sufficiente effettuare una in-visita e *contare* di volta in volta il numero di nodi visitati. Il k-esimo nodo visitato è il k-esimo nodo più piccolo nell'albero. Per poter implementare questo approccio è necessario assumere che il nostro linguaggio di programmazione ci permetta di passare argomenti per riferimento/indirizzo, in modo da poterne modificare il valore nelle chiamate ricorsive. Sotto tale assunzione, la seguente è una possibile soluzione al problema (Algoritmo 3). Nell'algoritmo 3 facciamo uso della sintassi C per indicare il passaggio per indirizzo dell'argomento k.

Algorithm 3: kthSmallestKey2(BST T, Int *k) → Int

```
if T = NULL then
|  return NA
else
|  res ← kthSmallestKey2(T.left,k)
|  *k = *k -1
|  if res ≠ NA then
|  |  return res
|  else if *k = 1 then
|  |  return T.key
|  else
|  |  return kthSmallestKey2(T.right,k)
|  end
end
```

L'algoritmo 3 effettua una in-visita dell'albero binario di ricerca che si interrompe (ricorsione a destra non effettuata) non appena viene visitato il k-esimo nodo. Nel caso peggiore (quando k è maggiore del numero di nodi nell'albero) la visita procede per tutti ed n i nodi dell'albero ed ha quindi un costo pari a O(n).

Come terza soluzione consideriamo il caso in cui l'albero non contenga informazioni aggiuntive (come abbiamo assunto per l'algoritmo 2) e che il linguaggio di programmazione ammetta solo funzioni con passaggio degli argomenti per valore (diversamente da quanto assunto per l'algoritmo 3). In questo caso, una possibile soluzione efficiente è quella di cercare il valore minimo e poi individuare iterativamente il nodo successivo per $k - 1$ volte (Algoritmo 4).

Individuare il minimo ed il successore di un nodo in un albero binario di ricerca costa nel caso pessimo O(h), dove h è l'altezza dell'albero. Effettuiamo sempre una ricerca del minimo e $k - 1$ ricerche del successore, quindi il costo computazionale nel caso pessimo dell'algoritmo 4 è O(kh).

Algorithm 4: kthSmallestKey3(BST T, Int k) → Int

```
node ← Min(T)
while k > 1 and node ≠ NULL do
|  node ← Successor(node)
end
if node = NULL then
|  return NA
else
|  return node.key
end

function Min(BST T) → BST
if T = NULL or T.left = NULL then
|  return T
else
|  return Min(T.left)
end

function Successor(BST T) → BST
if T = NULL then
|  return NULL
else if T.right ≠ NULL then
|  return Min(T.right)
else
|  P ← T.parent
|  while P ≠ NULL and T = P.right do
|  |  T = P
|  |  P = P.parent
|  end
|  return P
end
```

3. Un appassionato di televisione vuole guardare la quantità massima di trasmissioni nella medesima giornata. Una trasmissione televisiva è caratterizzata da un istante di inizio s ed un istante di fine e tale che $s < e$. Due diverse trasmissioni con, rispettivamente, istanti di inizio-fine $[s_1, e_1]$ e $[s_2, e_2]$ possono essere viste entrambe solo se non si sovrappongono i relativi intervalli di inizio-fine, ovvero $e_1 < s_2$ oppure $e_2 < s_1$. Ad esempio, date tre trasmissioni televisive con istanti di inizio-fine, il numero massimo di trasmissioni che possono essere viste è due: prima la trasmissione con inizio-fine $[2, 3]$, seguita dalla trasmissione con inizio-fine $[4, 7]$. Scrivere un algoritmo che dato un array di lunghezza n, che contiene coppie di numeri che identificano gli intervalli di inizio-fine di n possibili trasmissioni, restituisce il numero massimo di trasmissioni che è possibile vedere (ovvero, con intervalli di inizio-fine non sovrapposti).

   Soluzione. È possibile procedere secondo un algoritmo greedy, semplicemente selezionando le trasmissioni in ordine crescente di *fine* dell'intervallo, facendo attenzione a selezionare trasmissioni che non si sovrappongono. Ad esempio, date le tre trasmissione dell'esempio riportato nel testo, $[4, 7]$, $[1, 5]$, $[2, 3]$, si riordinano nel modo seguente: $[2, 3]$, $[1, 5]$, $[4, 7]$. Poi si selezionano le trasmissioni nel seguente ordine: prima la trasmissione con intervallo $[2, 3]$, poi si scarta la trasmissione con intervallo $[1, 5]$ in quanto ha inizio precedente all'istante 3, e poi si seleziona la trasmissione con intervallo $[4, 7]$.

   L'Algoritmo 5 prende in input gli intervalli delle n trasmissioni rappresentate tramite un array contenente n coppie di numeri. Si utilizzano i metodi $first$ e *second* per accedere al primo e al secondo elemento delle coppie. Il costo computazionale dell'algoritmo include il costo dell'ordinamento che, assumendo un algoritmo ottimale quale heapsort, risulta essere $O(n \log n)$, a cui si aggiunge il costo del ciclo che risulta essere $O(n)$. Complessivamente avremo quindi $T(n) = O(n \log n) + O(n) = O(n \log n)$.

Cont.

Algorithm 5: TRASMISSIONI(NUMBERPAIRS $T[1..n]$)

```
T.sort(second)       // Ordina gli intervalli in modo crescente rispetto al secondo elemento
numTrasmissioni ← 0; fineUltimaTrasmissione ← 0; i ← 1
while i ≤ n do
    if T[i].first > fineUltimaTrasmissione then
        // Si seleziona la i-esima trasmissione
        numTrasmissioni ← numTrasmissioni + 1
        fineUltimaTrasmissione ← T[i].second
    end
    i ← i + 1
end
return numTrasmissioni
```

4. Progettare un algoritmo che, dato un grafo orientato $G = (V, E)$, verifica se tale grafo contiene almeno un ciclo.

   Soluzione. È possibile procedere effettuando una visita in profondità, semplicemente verificando l'esistenza di almeno un arco all'indietro. A lezione si era discusso del fatto che questo si verifica se e solo se il grafo visitato contiene un ciclo. Più precisamente, si esegue una DFS che restituisce *true* se e solo se durante l'esecuzione della visita si incontra un arco che va da un vertice attualmente visitato a un vertice "grigio", ovvero già visitato ma non ancora chiuso. Tale soluzione è riportata come Algoritmo 6 e ha il medesimo costo della visita in profondità, ovvero $O(n + m)$, con n numero dei vertici e m numero degli archi, assumendo implementazione del grafo tramite liste di adiacenza.

Algorithm 6: ControllaCiclo(Graph (E, V)) $\rightarrow$ Boolean

```
// Inizializzazione marcatura dei vertici
for v ∈ V do
    v.mark ← white
end
// Esecuzione della DFS
for v ∈ V do
    if v.mark = white then
        if DFSvisit(v) then
            return true
        end
    end
end
// Se la DFS non ha incontrato archi all'indietro non vi sono cicli
return false

// DFS che restituisce true se e solo se si incontra un arco all'indietro
DFSvisit(Vertex u) → Boolean
u.mark ← gray
for v ∈ u.adjacents do
    if v.mark = white then
        if DFSvisit(v) then
            return true
        end
    else if v.mark = gray then
        return true
    end
end
// La DFS di u non ha trovato archi all'indietro
u.mark ← black
return false
```

The End.